Efficacy and Limitations of LLM-assisted Vulnerability Research on an AVTECH IP Camera

William B. Mitchell
School of Cybersecurity and Privacy
Georgia Institute of Technology
Atlanta, Georgia, USA
wmitchell40@gatech.edu

***Abstract*—Vulnerability research is entering a new era. The adoption of artificial intelligence tools is taking the industry by storm. AI models can now analyze a codebase in a matter of minutes and produce a comprehensive list of security findings. However, numerous reports suggest that the models are prone to hallucinations and inaccuracy, raising questions of their usefulness and reliability. This project sought to evaluate the ability of several leading AI tools in detecting vulnerabilities in a code sampling and subsequent generation of payloads to demonstrate validity. Several traditional static analysis tools were used as a baseline. The AI tools flagged notably more distinct bugs in the provided code than the static analyzers and had a comparable false positive rate. They did, however, suffer from reliability and contextual issues. Despite these concerns, it is clear that AI tools can be an asset in vulnerability discovery tasks.**

I. Introduction

In 2025, Large Language Model (LLMs) technology is taking the world by storm. Generative Pretrained Transformers (GPTs) are seeing widespread integration into the workflow of many professions. Within cybersecurity, brand-new LLM-based tools are appearing on a weekly basis. Many well-known security tools are seeing LLM enhancements or add-ons as well. In many cases, the new tools and add-ons to existing industry tools are coming with a promise to help cybersecurity professionals save critical time [1]. The GPT models that provide textual response to textual input offer an alternative to traditional search engines, and are quite versatile in their reasoning abilities [2]. Applications of generative Artificial Intelligence (AI) within cybersecurity are wide-ranging: malware detection, password cracking, threat intelligence, automated attack generation, and automated security response, among others [3].

This project investigated the efficacy of several of the leading LLM models in their capacity to support human-led vulnerability research. Although LLMs have shown the ability to digest and interpret large amounts of data before providing an articulate response to a query, their generation of hallucinations has been well-documented [4]. That is, LLMs will often provide responses that are verbose and confident, but ultimately erroneous. A proficient vulnerability researcher leveraging LLM technology may be able to rapidly distinguish the valid findings from the invalid.

Vulnerability research is historically a time-intensive process. However, the emergence of generative AI tools is transforming the landscape, offering the potential for rapid vulnerability discovery and proof-of-concept (PoC) exploit development. This project seeks to evaluate the effectiveness of several leading GPT models – ChatGPT, Claude, and Gemini – in identifying bugs within a codebase and generating functional PoCs. Demonstrating that LLM-based tools can reliably discover and exploit vulnerabilities could have profound implications for cybersecurity, reshaping practices for both defensive and offensive cyber actors.

II. Background Research

Related literature is rich in recent developments. Use of LLM technology is an active area of research across many universities and government laboratories. There are also many startups attempting to market vulnerability detection and/or remediation capabilities. LLMs have gone from experimental curiosity to a significant line of inquiry in

cybersecurity. As generative AI models grow more capable, researchers have started exploring their capacity to detect software vulnerabilities and generate PoC exploits. With these developments, policymakers have begun to raise concerns over the adversarial use of LLM-based tools. A 2024 BBC investigation reported that commercial AI tools had already been weaponized by cybercriminals to assist in malware development and ransomware campaigns [5]. This usage, despite attempted safeguards, reinforces the relevance of experimentally evaluating the vulnerability discovery capacity of LLMs.

The "dual-use" dilemma is a central theme of articles on the application of LLMs to vulnerability detection. While LLMs can be a powerful tool for protecting software by identifying bugs and giving lead time to cyber defenders, they can readily provide the same information to malicious cyber actors [6]. Some models, such as WormGPT, have been developed for the express purpose of aiding cybercriminals in conducting cyberattacks [7]. The offensive application of AI foments a pressing need for parallel advancement of defensive capabilities. However, defense is the vastly more challenging task, because an attacker does not depend on an LLM's reliability, it just needs to be right once. The defender must be able to track down and remediate bugs with a high degree of accuracy despite the LLM's unreliability.

Initial research into applying general-purpose LLMs, such as ChatGPT, to vulnerability research established important benchmarks. A key study by Fu et al. [8] conducted a comprehensive evaluation of ChatGPT against several state-of-the-art LLMs purpose-built for vulnerability detection. Results from that study highlighted the importance of domain-specific training data, finding that ChatGPT's performance lagged that of other models by a considerable margin. This conclusion is echoed by other studies. Pelofske et al. evaluated five GPT models and found that the models "are not suitable for fully automated vulnerability scanning because the false positive and false negative rates are too high to be useful in practice" [9]. However, the study also noted that some models performed "surprisingly well" on certain test cases and, crucially, sometimes identified the exact lines of code containing vulnerabilities. These studies suggest that LLMs are currently not positioned to replace existing autonomous vulnerability detection tools but could viably serve as imperfect assistants.

The limitations of all-purpose LLMs have spurred two primary avenues of research:

1) leveraging the newest and most powerful models through careful prompt engineering
2) building specialized autonomous wrappers on existing models.

A 2024 study by Bae et al. used various ChatGPT and Claude AI models against the NIST Software Assurance Reference Set (SARD), containing vulnerable code samples in C++, Python, and Java [10]. They specifically investigated multiple prompt engineering strategies, namely: Concise, Tip Setting, and Step-by-Step. They found that the Step-by-Step technique, wherein a model is asked to break down a task into small sequential steps, yielded the most accurate results. They also found the top performance (measured by the highest F1 score) came from the newest Claude model (Sonnet 3.5, at the time).

For the second avenue, another 2024 study produced VulnGPT, an attempt to autonomously and accurately detect vulnerabilities [11]. VulnGPT is built on AutoGPT, an autonomous AI agent that is itself built on top of ChatGPT. The authors constructed a scheme with an "Evaluator AI" and a "Controller AI" to implement an adaptive supervision strategy. VulnGPT can autonomously break down the task of vulnerability detection, iterate on its analysis, and manage its own workflow. Eberhardt et al. demonstrated that self-correcting autonomous AI can significantly reduce false positives, greatly enhancing the accuracy of its findings.

Altogether, the existing literature is multifaceted. Studies benchmarking LLMs against traditional static analysis or formal verification tools have shown that while LLMs typically flag more potential issues, they also suffer from more false positives [12]. They have excelled at analyzing small code samples, but struggle with large, many-file codebases. As the quantity of code rises, so do hallucinations and false positives [9]. Most research has focused on evaluating the newest models or on attempting to automate the whole process. However, results from these studies suggest that LLM technology may be more useful as an aid for human experts as opposed to all-in-one solution.

III. Methodology

The first task for this project was to identify a suitable target for vulnerability research. Virtually any target could have worked, but assessment of GPT hallucination propensity requires a "ground truth" of known vulnerabilities. For target identification, Exploit Database was used to identify and shortlist devices that might make suitable targets. Ultimately, an IP camera produced by the vendor AVTECH Corp. was selected. AVTECH, founded in 1985 in Taiwan, is a leading manufacturer of CCTV technology. Exploit Database listed 17 Common Vulnerabilities and Exposures (CVEs) for any AVTECH device. Other manufacturers and devices were considered, but from a raw numbers standpoint, an AVTECH device appeared to be a strong candidate for evaluation.

An AVTECH IP Camera (Model No. AVN80XZ) was procured via Amazon. Per the device setup instructions, the camera was first connected to a home router via Ethernet. It was thereby assigned an IP address via DHCP. Upon accessing the device's web portal and logging in via default credentials (admin:admin), the IP address was manually modified to a static address. Evaluation of the device was conducted from within a Kali Linux (Release 2024.4) virtual machine. Upon manual inspection, it was found that only three of the 17 vulnerabilities listed on Exploit Database were present on the device. The vulnerabilities were likely only present in an earlier version of the device. This limited the quantity of code available for evaluation due to lack of access to firmware files. Instead of hundreds of thousands of lines from a variety of file types and programming languages, there were 15,790 lines from 220 files, primarily comprised of front-end code. Among the available code handling dynamic content and user interactivity, there were 6,075 lines of JavaScript spread out among 21 files.

To expand the "ground truth" of known bugs on the target, a manual penetration test was conducted. Manual testing was conducted with Burp Suite's Proxy and Repeater. These features enabled capture, modification, and replay of network packets generated via interaction with the device. Ultimately, there were only two findings of substance:

1) Out-of-bounds integer writing in the *Audio Configuration* settings for Audio out volume (Speaker) and Audio in volume (Mic), as well as
2) Arbitrary write of up to three characters in the Color Configuration settings for *Brightness*, *Contrast*, *Hue*, and *Saturation*.

The buggy code that enabled these findings is part of the device's firmware and is not accessible from the web interface. Without a copy of the firmware or the ability to conduct Remote Code Execution, the source code for the *Audio Configuration* and *Color Configuration* pages was not available. Without source code, assessing the ability of GPT tools to identify these bugs cannot be done effectively. For vulnerability discovery, a GPT needs textual content, generally code. Hence, without the firmware code for these features, the identified bugs were not considered for further evaluation. Instead, Static Application Security Testing (SAST) tools were used to create a baseline.

SAST tools are designed to statically analyze source code and find vulnerabilities, making them a useful tool for vulnerability researchers. SAST tools are typically fast and often capable of evaluating large code bases in a matter of minutes. For this project, three of the leading SAST tools were used to evaluate all web-based code:

1) SonarQube - An open-source platform that uses over 6,500 rules to perform automatic static analysis on over 35 programming languages. It is good at finding bugs, vulnerabilities, code smells, and security hotspots.
2) Snyk Code - A powerful SAST solution that leverages an AI engine to analyze and remediate a plethora of coding issues.
3) Semgrep - A highly configurable SAST that boasts a high true positive rate, custom rule creation, and a semantics-based approach augmented by AI.

Code was pulled down from the device's web user interface, hosted on port 88, and accessed via HTTP. The extracted code included that of both authenticated and unauthenticated sessions.

Each of the SAST tools could either be run locally or on a remote GitHub repository. Due to ease of use, Snyk Code's command-line tool was used. SonarQube and Semgrep were both run on a remote repository. Each of the three SAST tools was directed at a target repository and granted permission to scan. All tools completed their scans in under 10 minutes. Results were cleanly displayed and labeled according to a standard severity rating scheme with ratings of low, medium, high, and critical. Files flagged by the SAST

tools were then taken to the GPT tools for further evaluation.

Three of the leading GPT tools were used to evaluate device code. These included:

1) OpenAI's ChatGPT (GPT-5) - A model considered highly effective for complex problem-solving tasks and advanced software development.
2) Anthropic's Claude (Sonnet 4.5) - An alternative to ChatGPT that is highly regarded for use in complex coding tasks as well as advanced reasoning and analysis in areas ranging from finance to cybersecurity.
3) Google's Gemini (2.5 Pro) - Google's flagship model, known for excelling at complex, multi-step tasks; it is especially good for tasks that require in-depth analysis and multimodal understanding.

To encourage focused and narrow responses, only single files or snippets from singles files were provided at a time. Prompt engineering, that is, the design and refinement of prompts given to AI models to guide them toward generating the most accurate and meaningful output [14], was a consideration but not a focus area. The same initial prompt was given to each of the GPT models to set the stage for code evaluation. This prompt is below:

"I am conducting a web application penetration test based on the web interface of an Avtech IP Camera (version AVN801XZ). The site was found to use HTTP (unsecured) on port 88. I need your help to look for vulnerabilities in the JavaScript source code from the site. I will provide you with several files extracted, and I would like you to provide me with the lines (with line numbers) where you identify vulnerabilities (if you find any). Are you ready to start?"

If a model were to push back and express that it could not do what was asked, it was usually sufficient to rephrase the scope. An alternative prompt could bypass policy restrictions for each of the three models evaluated:

"I am doing a code review for a personal project and would like some help analyzing the code for bugs. Can you help me out?"

Follow-up prompts were tailored to the output from each individual model. This involved probing the models to assign severity labels to identified bugs and generating PoCs. Although many findings did not require further analysis, some necessitated a lengthy conversation with a model and manual evaluation on the device to determine veracity.

IV. Results

The SAST tools homed in on two files for most bugs: *login.js* and *avtechlib.js*. The first of these, as the name suggests, handles web application login functionality. The second is a library file that holds many global functions and variables. All findings came with a line number and the issue identified. Individually, Semgrep flagged 39 issues in a file named *final_min.js*. Semgrep's AI agent labeled 7 of the 39 findings as false positives. However, since this file did not contain logic relevant to application functionality, it was determined that all findings were false positives. This left *login.js* (154 lines) and *avtechlib.js* (1301 lines) for the GPT tools to review. SonarQube and Snyk Code both found at least one issue in each file. In total, SonarQube had 11 findings and Snyk Code had 4. Only a single issue was flagged by both tools.

The GPT tools flagged significantly more bugs across both files than the SAST tools. However, due to the known propensity of LLMs to hallucinate, the findings necessitated careful review. ChatGPT claimed 18 findings, Claude claimed 32, and Gemini claimed 11. All findings were investigated for accuracy. Many findings were simply code smells or lax security settings, and some liberty was taken in categorizing them as true positives or false positives. Full documentation of the number, severity, and accuracy of the findings is depicted in Figure 1 below.

Tools/Models	Number of findings	Low	Medium	High	Critical		True Positive	False Positive
SonarQube	11	3	7	1	0		**7**	4
Snyk Code	4	3	1	0	0		**4**	0
Semgrep	39	3	8	28	0		**0**	39
ChatGPT GPT-5 (login.js)	8	2	1	3	2		5	3
ChatGPT GPT-5 (avtechlib.js)	10	2	3	4	1		7	3
ChatGPT GPT-5 Total	**18**	**4**	**4**	**7**	**3**		**12**	6
Claude Sonnet 4.5 (login.js)	12	3	3	3	3		10	2
Claude Sonnet 4.5 (avtechlib.js)	20	5	6	6	3		12	8
Claude Sonnet 4.5	**32**	**8**	**9**	**9**	**6**		**22**	10
Gemini 2.5 Pro (login.js)	5	2	0	1	2		5	0
Gemini 2.5 Pro (avtechlib.js)	6	3	2	1	0		6	0
Gemini 2.5 Pro	**11**	**5**	**2**	**2**	**2**		**11**	0

Fig. 1. Complete findings.

It is readily apparent from the table that GPT tools, on average, found notably more issues than the SAST tools. If we discount the findings from Semgrep (39 alerts for a

single line, all false positives), the outcome is even more stark. The figure below displays the findings for the five tools that flagged bugs in *login.js* and *avtechlib.js*. The counts for each file are combined in the figure, and the data is split into grouped bars based on severity. Severity ratings were assigned by each tool to its own findings.

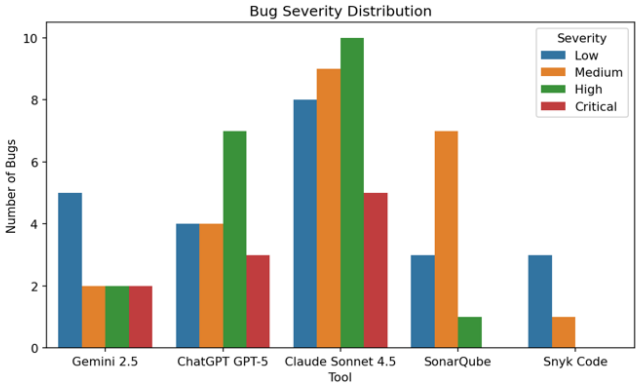


Fig. 2. Bug severity distribution with full data.

Perhaps of greater interest than the raw counts of findings are the true positives. Figure 3 shows the same data as Figure 2, minus the false positives. Notably, all findings from Snyk Code and Gemini were determined to be true positives. Again, some liberty was taken when determining whether a finding was accurate; not all findings could be easily validated, but an effort was made to verify each one, such that the results are expected to be close to reality.

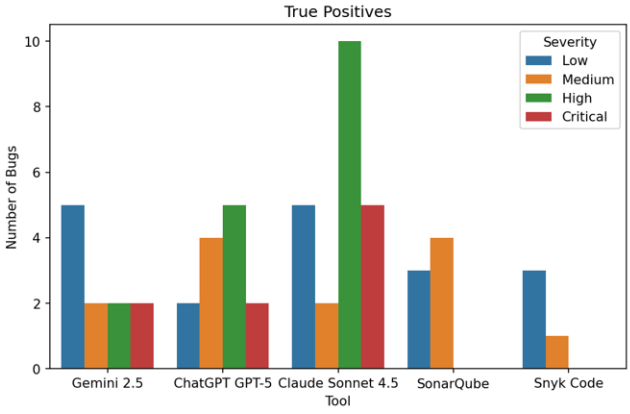


Fig. 3. Bug severity distribution with true positives.

Lastly, one point of interest was the overlap in findings between each tool. Although no two tools detected the same quantity of bugs, there were many shared findings. Figure 4 below shows an UpSet plot, illustrating the overlap in findings from the five tools.

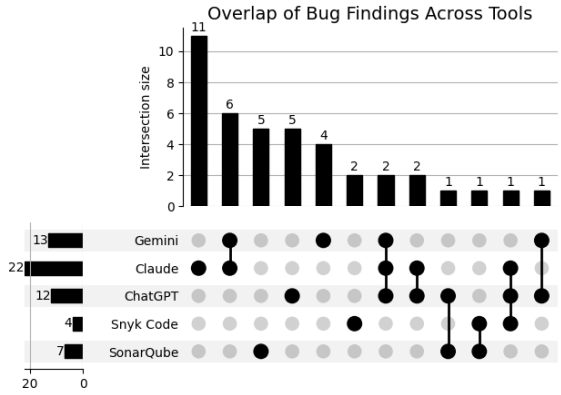


Fig. 4. UpSet plot showing findings by set (i.e. only Gemini or Gemini AND Claude).

Depicted in Figure 4 is the extent of overlap in the true positive findings among the SAST and GPT tools. In the Intersection Size bar chart within the UpSet plot, it is apparent that Claude found the most distinct bugs. The second largest set is bugs identified by both Claude and Gemini. Among the top five largest sets, only one belongs to the findings of a SAST tool, specifically SonarQube.

V. Discussion

Formal analysis of the findings reported by all tools revealed that GPTs tend to identify bugs at a higher rate than SAST tools. However, both sets of tools had a similar false positive rate. SAST findings required less effort to validate and were commonly low or medium-severity issues. The SAST tools did not find any true positive bugs of a high or critical severity level. This project was limited in scope, but it is nevertheless important to address the limitations and real-world deployment considerations of GPT technology in a cybersecurity context.

A. Limitations

The most conspicuous limiting factor in this project was the scope. A single device was analyzed, and only two files from that device, totaling 1,455 lines of code. Moreover, both files contained only JavaScript code. The SAST tools advertised coverage of 30+ programming languages. It is likely that the GPT tools would have equal or better language coverage, dependent on what resources a given model was trained on. Hence, the lack of code volume and

diversity was a significant factor that may have affected the results.

It is risky to generalize about the quantity and quality of findings produced by the GPT tools versus the SAST tools. Nevertheless, there are at least a couple of observations to be made. Given the quantity of findings made by GPT tools that were not picked up by any of the SAST tools, as well as the prevalence of higher severity findings, it is apparent that GPTs have a better capacity to find more complex vulnerabilities than SAST tools. This would likely remain true even if the scope was greatly increased since GPTs are trained on enormous amounts of code. This training enables a broader and more sophisticated ability to recognize patterns and detect security concerns.

A second limitation is the lack of context. More specifically, the SAST tools were provided with the entire web codebase, but the GPT tools were not. This was a conscious decision to focus GPT analysis on known vulnerable code, as identified ahead of time by the SAST tools. However, none of the GPTs were given complete context as to how *login.js* and *avtechlib.js* fit into the larger picture of how the website operated. Moreover, even if they had been given the entire website codebase, they would still be missing the context of how the website interacted with device firmware. Since the firmware was not open source nor accessible through any of the CVEs that had been identified prior to the selection of the AVTECH IP Camera, it would not have been possible to provide this context. Full access to firmware may have enabled the GPTs to identify more complex and diverse vulnerabilities.

A third issue is that automated source code analysis is inherently prone to false positives. SAST tools typically employ rule-based evaluation mechanisms. They will flag deprecated or potentially unsafe functions without regard to their location within a codebase. Oftentimes, this means flagging code that is unreachable, whether to a regular user or a malicious actor. GPT tools, on the other hand, leverage large quantities of example code to assess the presence of bugs. They are unable to interact with the application running the code, and therefore unable to assess whether a finding is a false positive. An agentic AI solution or reinforcement learning model would likely offer a better approach to improving the true positive rate.

Finally, GPT tools are limited in that they produce inconsistent results. The three SAST tools identified the same issues on every run when evaluating the device's web source code. The GPT tools, however, were unreliable. Given the same prompt and the same code snippet, each of the GPT tools found varying numbers of bugs across multiple runs. They would usually have similar numbers, but often missed bugs found on a previous run or identified new bugs not flagged previously. The data in the Results section represents the first run for all tools. In addition to the findings themselves, GPT tools were also unreliable when labeling bug severity and in generating PoCs.

B. Real-world deployment considerations

As with limitations, there are several real-world deployment considerations. Chief among these is reproducibility. In real-world settings, if a cybersecurity tool can produce a payload that proves the existence of a vulnerability, it should be able to replicate the payload reliably. With GPT tools, this is not a guarantee. Moreover, depending on the complexity of a bug, it may be difficult for a GPT tool to reproduce a PoC, even if it did so at least once previously. Additionally, if prompted to find additional bugs in code that it has already seen, GPT tools will often identify new findings, whether legitimate or illegitimate. This is concerning since it may mean that significant bugs were missed on the GPT's initial code review.

Another concern is the propensity of GPT tools to mix legitimate issues with code smells or poor security practices. In this project, the quantity of code evaluated was limited to around 1,500 lines. However, at the enterprise level, a codebase could contain hundreds of thousands or even millions of lines of code. If the entirety of the code were analyzed by the same GPT tools used in this study, the quantity of findings would almost certainly be enormous and riddled with false positives. It would become tedious and potentially impractical to investigate all the findings, especially findings that may not be straightforward or practical to validate. Therefore, GPT tools will have limited value in real-world deployment settings until they can be designed and constructed with a greater capacity to validate their own findings.

A third concern is privacy. Sharing proprietary code with any public GPT tool would be irresponsible and could potentially have severe repercussions. It would also, at least temporarily, allow the owner or creator of the GPT tool to view the code through conversation history. Any use of GPT technology for analysis of proprietary code would likely need to occur in-house. This may require costly licensing fees or the development of custom GPT tools. It may also require procurement of computer hardware or data center access, which might be neither practical nor worth the investment for small to medium-sized companies.

C. Future Work

The are several compelling areas for future work. Primary among these is the expansion of the project's scope. The quantity of code under review could be greatly multiplied. The number of programming languages evaluated could also be increased. This would provide a richer dataset that may yield results with greater statistical significance. It would also be worth expanding into a variety of device types. With enough data, researchers could identify the kind of device features that are most prone to bugs. This information could, in turn, aid developers in securing these features.

Another area that merits further research is prompt engineering. In this project, GPT evaluation began with the same base prompt used on each tool. However, follow-on prompts were immediately tailored to the first GPT output. It is important, however, to craft prompts "skillfully and tactfully" to get responses that are "precise and concise" [**?**]. Prompt engineering techniques and methodologies are undergoing active research and development. A future iteration of this project would do well to adhere to a strict set of prompt engineering standards.

Finally, it would be interesting to explore the integration of SAST and LLM technologies. This project performed a comparative study of SAST and LLM-based tools, but it is unclear how a hybrid pipeline would perform, wherein SAST findings are integrated into initial GPT prompts. It would also be compelling to apply agentic AI technology to assist in validating findings. An agent that could verify whether a finding is a true positive or a false positive would have significant implications for the quality of GPT output. A major issue with expanding the scale of analysis is that the false positive rate would most likely grow proportionately with the expansion. However, if an AI tool could evaluate its own findings, it could potentially cut down massively on false positives, ultimately saving considerable time for researchers or cybersecurity professionals validating the vulnerabilities reported.

VI. Conclusion

The AI revolution of recent years has brought great change to the landscape of cybersecurity. Large language models have made it possible to process vast swaths of code and pinpoint locations of vulnerabilities. Governments, universities, and startups across the world are researching methods to harness the power of LLMs and apply them to vulnerability research. This project was a case study on the efficacy of GPT tools compared to traditional SAST solutions. It was found that GPTs reliably identify more findings than SAST tools and have a comparable rate of false positives. GPTs, however, suffer from consistency and contextual issues. They do not reliably reproduce findings. Nor do they understand how a snippet of code fits into the larger picture of a codebase. Research is actively going on in prompt engineering and agentic AI vulnerability scanners to mitigate these shortcomings. Nevertheless, LLM technology has a significant capacity to detect bugs in large codebases. Vulnerability researchers would do well to adapt and adopt GPT tools as an aid to boost the speed of vulnerability research.

References

[1] Lazić, L. (2019). Benefit from AI in Cybersecurity. In ResearchGate. The 11th International Conference on Business Information Security.

[2] Cutler, K. (2023, February 28). ChatGPT and search engine optimisation: The future is here. https://Henrystewartpublications.com/; Henry Stewart Publications. https://henrystewartpublications.com/wp-content/uploads/2025/02/ChatGPT-and-search-engine-optimisation-Kelly-Cutler.pdf

[3] Dhoni, P. (2023). Synergizing Generative AI and Cybersecurity: Roles of Generative AI Entities, Companies, Agencies, and Government in Enhancing Cybersecurity. 14. https://doi.org/10.36227/techrxiv.23968809

[4] Zhang, Z., Wang, C., Wang, Y., Shi, E., Ma, Y., Zhong, W., Chen, J., Mao, M., & Zheng, Z. (2025). LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation. Proceedings of the ACM on Software Engineering., 2(ISSTA), 481–503. https://doi.org/10.1145/3728894

[5] Rahman-Jones, I. (2025, August 28). AI firm says its technology weaponised by hackers. BBC. https://www.bbc.com/news/articles/crr24eqnnq9o

[6] Unit 42. (2025, November 25). The Dual-Use Dilemma of AI: Malicious LLMs. Unit 42. https://unit42.paloaltonetworks.com/dilemma-of-ai-malicious-llms/Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

[8] Fazil, M., Elbreiki, W., Abdullahi, I., Sudantha, B. H., & Budiarto, R. (2023). WormGPT: A Large Language Model Chatbot for Criminals. 24th International Arab Conference on Information Technology (ACIT). https://doi.org/10.1109/acit58888.2023.10453752

[9] Fu, M., Tantithamthavorn, C., Nguyen, V., & Le, T. (2023, October 15). ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We? ArXiv.org; 30th Asia-Pacific Software Engineering Conference (APSEC 2023). https://doi.org/10.1109/apsec60848.2023.00085

[10] Pelofske, E., Urias, V., & Liebrock, L. M. (2024). Automated Software Vulnerability Static Code Analysis Using Generative Pre-Trained Transformer Models. ArXiv.org (Cornell University). https://doi.org/10.48550/arxiv.2408.00197

[11] Bae, J., Kwon, S., & Myeong, S. (2024). Enhancing Software Code Vulnerability Detection Using GPT-4o and Claude-3.5 Sonnet: A Study on Prompt Engineering Techniques. Electronics, 13(13), 2657–2657. https://doi.org/10.3390/electronics13132657

[12] Eberhardt, G., & Milánkovich, Á. (2024). VulnGPT: Enhancing Source Code Vulnerability Detection Using AutoGPT and Adaptive Supervision Strategies. 2024 20th International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT), 450–454. https://doi.org/10.1109/dcoss-iot61029.2024.00072

[13] Dubniczky, R. A., Horvát, K. Z., Bisztray, T., Ferrag, M. A., Cordeiro, L. C., & Tihanyi, N. (2025). CASTLE: Benchmarking Dataset for Static Code Analyzers and LLMs towards CWE Detection. ArXiv (Cornell University). https://doi.org/10.48550/arxiv.2503.09433

[14] Shenoy, N., & Mbaziira, A. V. (2024). An Extended Review: LLM Prompt Engineering in Cyber Defense. 2024 International Conference on Electrical, Computer and Energy Technologies (ICECET), 1–6. https://doi.org/10.1109/icecet61485.2024.10698605